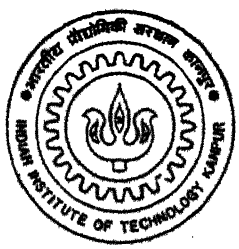
Robust Geometric Computations

by
PARAG SHRIKANT KALE



DEPARTMENT OF COMUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
MAY, 1995

Robust Geometric Computations

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

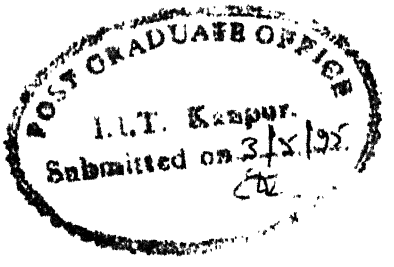
MASTER OF TECHNOLOGY

by
Parag Shrikant Kale

to the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY,
KANPUR
May, 1995

Certificate

This is to certify that the work contained in this thesis entitled **Robust Geometric Computations** by **Parag Shrikant Kale**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



Dr. Asish Mukhopadhyay
Associate Professor,
Department of Computer
Science and Engineering
IIT, Kanpur

— to

Monty, my cat,

for being a better friend than any human.

Acknowledgements

At the onset I express my deepest gratitude to my parents for reasons more than one. I am happy that their hardships are not otiose.

I am extremely grateful to my thesis supervisor Dr.Asish Mukhopadhyay for allowing me to work with him and for introducing me to one of the most interesting aspects of geometric modeling — robustness. I envy his resilent composure for tolerating as obstreperous a student as me for so long.

I also thank Dr. Sanjeev Saxena for being kind and affectionate to me every time I approached him for some reason or the other.

I am indebted for my entire lifetime to TAN for without him I would have never got my degree in this convocation. During the trying times in the last few days I reminisced Srin, RS and Kuntu who used to come to my aid even before I called them.

Others who tried to make life easy in this barren land were Abhijit, Devendra, Anand, Neeraj, Nilesh, Gujral, Pavan, Ajay and Vishwas. I thank Uma bhabhi for all the delicacies she prepared for me.

Parag.

Abstract

Robustness of geometric algorithms is one of the outstanding problems faced by the programmers in computational geometry and related areas. A comprehensive review of the current status of the robustness problem is presented here. One of the main reasons for the failure of geometric programs implemented by the programmers is the inconsistency between numerical and symbolic data. We suggest a method for geometrically consistent maintenance of numerical data. The proposed method ensures that an approximated model exists for the geometric relations derived using the numerical data. We represent the uncertainty in determining a point as a product of the uncertainty in determining each of its coordinates. This interpretation of the uncertainty has certain advantages over the current interpretation where the Euclidean metric is used to measure the uncertainty.

Contents

List of Figures

Chapter 1

Introduction

The lack of a general framework to address the robustness of geometric algorithms has given rise to many solutions, with different names but having similar ideas. A comprehensive review of the state of the art on robustness is presented here. One of the major reasons for the failure of geometric algorithms is the inconsistency between the numerical and the symbolic data that is used to define the geometric object. We give a method which exploits the uncertainty in the numerical data to obtain a refined numerical data which is consistent with the symbolic data. Once the numerical data is refined we can use it to deduce consistent geometric relations between the various geometric objects. This technique is computationally expensive and so we emphasize that the complexity of the program should be related with the *correctness* and *accuracy* of the solution that it produces. This is in marked contrast with the asymptotic time complexity, and it is more *real.*

The point location problem in two dimensions is used as a means to explore the robustness issue. Given a planar subdivision i.e. a division of the plane into connected planar regions we are required to locate a query point in one of the elements of the subdivision. Here elements are the vertices, the edges and the regions of the subdivision. This problem has been completely solved in the continuous domain [EGS86, PS85]. However, with approximate arithmetic and uncertain data, a satisfactory solution is still elusive. Here we suggest techniques to overcome some problems .

1.1 Robustness Of Geometric Algorithms

The implementation of robust and efficient algorithms for computational geometry and related areas is a far cry from reality. Though the issue of robustness has been widely appreciated [Ch94] the theoretical framework to tackle it has not yet evolved. The current status of this issue is best explained by quoting Milenkovic—

> There are two reasons for the fact that the current theory of computational geometry does not ordinarily address the issue of robustness. First, current theory is based on an analysis of asymptotic cost, and, for any specific geometric construction, the cost of using unlimited-precision rational arithmetic (or whatever is necessary to implement the algorithm without rounding) is only a constant factor greater than using rounded floating-point arithmetic. Second, it is assumed (presumably by those who have never implemented a geometric algorithm), that round-off error pose only minor, easily surmountable difficulties. Experience teaches us that this is a false assumption.

Interested readers are kindly referred to [Mil89, GY86, FBZ93, Mil93, Mil88, SS84] for understanding, how erroneous the use of *ad hoc* tolerances can be.

Few concepts need to be introduced at this stage for better understanding of the problem [HHK88, Hof89]. A geometric object has a *representation* and a *model*. A *representation* has symbolic data like adjacency and incidence relations and finite-precision numerical data. For example in the representation of a polygon we can specify the edges as links between vertices. Then the numerical data is the coordinates of the vertices and the symbolic data consists of the links. The representation has a *model* in the Euclidean space if there exists an object which satisfies the symbolic data of the representation. The numerical data in the representation can be inaccurate and the symbolic data need not confirm to any rule, but a model exists only if there are numerical values for the geometric object that make the geometric relations feasible. Since the model is in the continuous domain it may require infinite-precision for its numerical data.

Thus the origin of the problem is clear—finite precision numbers (bounded size integers, fixed-point and floating-point numbers). Almost all algorithms are designed and proved to be correct in the continuous domain, for sheer analytic convenience. However real machines provide a discrete domain. This fact has grave consequences, for example we cannot model an equilateral triangle in a computer if we store the coordinates of its vertices. In 2D, out of the six values which represent the coordinates of a triangle at least one is irrational if the triangle is equilateral. The assumption that the discrete domain offered by the machine is as good as the continuous one does lead to acceptable results in some cases but fails miserably in many cases of practical importance.

We now have to look at approximate models for the representations and this brings error analysis in the picture. The role of errors in arithmetic computations is fairly well understood but little can be said about their geometric manifestations. Many times decisions are made in a geometric algorithm, depending on the sign of an expression representing a geometric predicate [Sto88]. The value of the expression may be numerically stable but not the sign. Informally, an arithmetic computation is numerically stable if a small perturbation in the input causes a relatively small perturbation in the output. A small change in the input can change the sign and so finite precision numbers, due to rounding errors, are bound to give wrong results.

The preceding discussion is sufficient to state the notion of robust algorithms.

- **Feasibility :**

  The representations of the input and output of an algorithm should have exact models. The exactness of models is a very strict condition and in fixed precision we have to settle for approximate models.

  In this paragraph we further elucidate the concepts introduced in the preceding one. It is possible that in the course of computation the algorithm infers that two lines intersect at more than one point [Mil93]. This is an unrealizable situation for straight lines. It is also possible that the representation generated by the algorithm violates some fundamental theorem in geometry, like the Pappus'

theorem, and so has no model in the Euclidean space [HHK88].

A concept closely related to feasibility of algorithms is the *correctness* of binary geometric computation, for example the computation of the intersection of two simple polygons. The correctness of a geometric procedure is no guarantee that a cascade of such procedures, is also correct [HHK88].

- **Accuracy :**

  This aspect is the consequence of the approximation of models. The error in the input and the output models with respect to their corresponding exact models should be bounded above. This error is also called the *distance* between the exact and inexact models. This distance should be independent of the size of the input.

1.2 Degeneracy

Robustness has another aspect—the problems due to the assumption of simple input. An algorithm may begin with the assumption that there are no vertical edges or that no four points are co-circular, in the input data. These exceptions are the degenerate cases. An input free from degenerate cases is termed as simple or said to be in general position.

Degeneracy has two associated problems, one of detecting it and another of handling it. When an arithmetic expression representing a geometric predicate evaluates to zero a degeneracy is detected. The input is then symbolically perturbed to get a sign, consistent with the previous computation. The perturbation is infinitesimal and symbolic. This technique has been used by many authors;[EM90, EJ91, EC92] tackle the perturbation of determinants and [Yap88] handles polynomials. Knuth [Knu92] has used this technique to prove the correctness of his algorithms for convex hull and Delaunay triangulation.

Yet symbolic perturbation is not a panacea. Arguments against it can be found in [BMS]. In [BMS] exact computation is used to detect degeneracy but degeneracy

is handled directly. The cost of symbolic perturbation is exorbitant because it needs exact computation. Moreover, rarely is the input data exact. These reasons are sufficient to render this technique impractical. Furthermore, such singularities may be a part of the design specification and removing them will make the geometric model meaningless.

1.3 Thesis Organization

The second chapter presents a logical rather than a chronological evolution of techniques to achieve robustness. The third chapter explains the point location problem and discusses the implementation of the algorithm in [EGS86]. In the fourth chapter we propose a method that achieves consistency between the numerical and the symbolic data. Also a metric to quantify the incidences is suggested which can be used for the point location problem. The appendix points out a small correction to the algorithm mentioned in [EGS86]. The bibliography contains the references cited in the thesis.

Chapter 2

Robust Geometric Computations - A Retrospective

The concept of robust geometric algorithms was introduced in the previous chapter. By now it is fairly clear that the robustness issue is a geometric manifestation of the finite precision problem. Inaccurate numerical data gives rise to inconsistency between the numerical data and the symbolic data. It was also stated that the most favored heuristic of using *ad hoc* tolerances, fails miserably in many cases of practical importance. Here we look at various methods that have been used to get robust solutions to a plethora of geometric problems.

2.1 Increasing Numerical Accuracy

By increasing the accuracy of the computations, consistent relations amongst the objects can be deduced. However when the input is uncertain this approach is inadequate to achieve robustness. This technique can be effectively used to detect ill conditioned cases, which can be handled later on by geometric reasoning.

2.1.1 Exact Computation

It is believed that robustness will be a non-issue with the use of exact computation [Yap]. Here, exact computation does not demand infinite precision but sufficient precision. Numerical quantities may be represented as rational numbers to get better results than the use of pure integers. Vaguely, in this paradigm we don't use any kind of approximation and no errors are ever committed during the computation. Due to this, classical geometric concepts are preserved and implementation of geometric algorithms becomes a relatively simple issue. Moreover, there are certain problems like geometric theorem proving, checking geometric conjectures, symbolic perturbation, which work only with exact computation.

Chee-Yap describes a generic software support required to implement algorithms exactly. Problems that don't exploit the full power of arithmetic operations i.e. use only the four basic arithmetic operations $+$, $-$, $\times$, *and* $/$ and have rational input are called *rational* problems. Linear programming, construction of hyperplane arrangements, point location, are some instances of rational problems. An algorithm is said to be rational *bounded* degree, if there is a constant D such that the intermediate computation involves rational numbers of size at most $Dn + O(1)$ when the input instance to the problem has rational numbers of size at most n. There are not many natural problems in traditional computational geometry that are rational but not bounded degree. Only when the problem is rational bounded degree the storage requirements can be predicted. Delaunay triangulation and planar point location are bounded degree problems.

We turn our attention to the efficiency of exact computation. If every computation in the algorithm is replaced by its exact counterpart than the algorithm slows down by more than 10000 times [KLN91]. Therefore efficient ways to evaluate the geometric predicates become imperative. The evaluation of many geometric predicates is tantamount to finding the sign of a determinant e.g. the orientation predicate. In 2D this predicate tells, whether three points are collinear, anti-clockwise oriented, or clockwise oriented and it is represented by a 3×3 determinant. The sign of a determinant is so often used to find the value of geometric predicate that we are obliged

to search efficient ways to evaluate determinants.

2.1.2 Efficient Determinant Evaluation

Clarkson [Cla92] gives an algorithm to compute the sign of a determinant for a $d \times d$ matrix with b-bit entries. It requires $O(d^3)$ arithmetic operations on numbers of bit-length $O(b+d)$. A fair operation count for the algorithm is not known. Rosenberger, in his doctoral thesis [Ros90] shows that the conventional methods to evaluate determinants viz. expansion by minors, Gauss elimination, have exponential time complexities. This increase is due to the fact that the size of the intermediate results grows at every stage of evaluation and the processor can operate on only two words at a time. Thus to multiply two, 2 word numbers we need 4 one word multiplications and 3 one word additions along with bit shift operations. Csanky gave a polynomial time algorithm for determinant evaluation and can be found in [Ros90]. [FW93] show that, the dynamic programming approach also has an exponential time complexity.

2.1.3 Modular Arithmetic

The count of elementary (or one word) operations can be reduced by reducing the size of intermediate results. Modular arithmetic can be used to achieve this goal and it is given in [Knu69, AHU74]. In modular arithmetic instead of representing an integer by a fixed-radix notation we represent the integer by its residues modulo a set of pairwise relatively prime integers. But the reconstruction of the sign of the determinant from the residues and in some cases operations like the modular inverse are expensive. An optimization in the size of the residues and the size of the primes has to be achieved for good results. If the size of the residues grows than we are back to long integers and if the size of the primes is increased than the reconstruction of the sign of the determinant becomes expensive.

2.1.4 Disadvantages of exact computation

Before proceeding further we digress to consider an important shortcoming of exact arithmetic (integer, rational, modular) and that is the assumption of exact input. This assumption is almost always false when the input is generated by another program employing floating-point numbers. Also, the input may be given in an implicit form in the continuous domain. Exact computation restricts all representable numbers to a grid if the bit lengths are bounded, which is the usual case. Adjusting the geometric entity so that it can be represented by these grid points leads to an exact solution of an approximate problem. Moreover with finite bits in the representation of numbers we cannot model such simple entities as the equilateral triangle. More examples requiring irrational numbers are given in [Hof89].

Sugihara [Sug89] presents an experimental study on the approximation of continuously distributed geometric objects by equations with integer coefficients. The problem he considers is, given an equation of a geometric object with real coefficients to find a set of integer coefficients that gives a good approximation to the original equation. The problem of maintaining the integrity of a geometric object when subjected to rotations is studied by Canny *et al.* [CDR92]. They give an efficient method to find rational approximation to the sine of the given angle. Greene and Yao [GY86] demonstrate an approach to modify an algorithm in the continuous domain so that it works even in the discrete domain, with the aid of the segment intersection problem. The idea here is to confine the lines in the continuous domain to an envelope of points on the screen which also includes the raster points displayed by most line drawing algorithms. This is accomplished without losing many nice topological properties of the continuous domain. But this approach is efficient only for display oriented output. Efficient methods for finding rational approximations to real numbers can be found in [Sug89, Hof89, CDR92].

2.1.5 Adaptive Precision

We return to the cost of naive implementation of exact computation. Do we need to evaluate every predicate in the program exactly ? If the instance of the predicate is not near-degenerate i.e it is not ill conditioned we can use floating-point arithmetic to get the truth value of that predicate. This forms the basis of adaptive precision. We resort to exact computation only when the calculated result is uncertain and so we cannot rely on it. This technique is reported in [KLN91, FW93]. The concept of adaptive precision is vaguely reminiscent of interval analysis. In interval analysis we replace the evaluation of an expression by the evaluation of guaranteed upper and lower bounds on its value. If the bounding interval does not contain zero than the sign is correctly known otherwise higher precision evaluation is necessary.

It was experimentally found that the chance of getting an uncertain result while evaluating a 3×3 determinant is considerably less, less than one in a million. The entries of the determinant were randomly generated single precision numbers between 0 and 1, using a linear, congruential, pseudo-random number generator. After that near-degenerate cases were generated i.e. if the determinant is evaluated with single precision then the sign is uncertain. These determinants were than evaluated using double precision and it was found that less than one percent determinants had an uncertain sign. This clearly shows that in a practical situation we rarely need higher than double precision and also the probability of using still higher precision is negligible. Thus the dynamic extension of precision seems to be a feasible solution which is indeed true of rational bounded algorithms.

2.1.6 Static Error Analysis

Finding the uncertainty interval requires error analysis and there is no unique way to calculate the uncertainty interval. The most preliminary technique is given in [Knu69] and still forms the basis of error analysis in many papers. In the best case the bounds on the errors are pessimistic and in the worst case they can prove to be gross miscalculations. The problem with these static error estimates is that the

assumption that the relative error model for approximate arithmetic holds at every stage of error estimation which may not be true. This case arises when we have subtraction in one of the intermediate stages. The subtraction of two, nearly equal numbers causes a significant loss of accuracy and the error estimates after this stage go haywire.

2.1.7 Dynamic Error Computation

The shortfalls of static error analysis brings in the concept of dynamic error computation. Vignes and La Porte introduced the permutation-perturbation method for error analysis, and was applied by Dobkin and Silver [DS88] for, the general, iterative, intersection problem. They experimented with the idea of calculating the number of significant digits in the computed result at every stage, and if the accuracy is less than desired, backtrack to the previous stage and start the computation at a higher precision. The issues that arise out of naive backtracking and an account of the experiments can be found in that paper, but it provides no concrete answers to the problem of robustness.

Masotti [Mas93] suggests that floating-point numbers should be represented by means of a data structure, holding the the value along with the relative and absolute errors. This technique seems to hold promise, despite the fact that a similar technique was deemed to be inefficient in [FW93]. Masotti shows that by monitoring the estimated relative error during the computation the validity of the results can be ensured. That paper reports statistics about the classical pentagon in-out problem which was tackled in [DS88]. The error estimates ensure numerical stability and enable detection of ill conditioned cases. Dynamic extension of precision under the control of error estimates is advocated in that paper. To speed up computation, in wake of the new data structure for floating-point numbers Masotti proposes a hardware design for a special floating-point processor.

2.1.8 Other Attempts

Ottmann *et al.* [OTU87] show that the *lsba* evaluation of an expression can be used to obtain a robust algorithm for the segment intersection problem. Here *lsba* stands for *last significant bit accurate* i.e. as accurate as the machine precision. The idea is to ensure best possible results for floating-point operations. That paper however assumes exact input and needs software support for the *lsba* evaluation of scalar product of two vectors. It is required to have access not just to the computed data but also to the data from which it is computed. This technique does not seem to be easily applicable to other problems.

2.2 Introducing Error In The Arithmetic Model

Here we begin with the assumption that the computation and sometimes even the input is inaccurate. Thus the crux of this approach is to accept *reality* rather than simulate *exactness*. The symbolic data presides over numerical data. If the input is inaccurate then the errors should be bounded. The computations are error prone due to rounding of intermediate results. We assume that we are not allowed to look into the internal representation of the floating-point numbers and manipulate them to gain precision. The problem boils down to making consistent decisions in the presence of error.

To find the error, analysis is usually done on the basis of the relative error model given in [Knu69]. As stated in [Knu69] there is considerable loss of significance when two almost equal numbers are subtracted. If this insignificant result is used in further computations then the relative error model is no longer valid. But this fact is condoned in almost every paper. Another approach to error analysis, favored by Milenkovic, is the use of absolute errors. In this case we use fixed-point numbers i.e. numbers that are rounded to the size of the mantissa, but have no exponent. We thus use the maximum error that can occur in the computation.

2.2.1 Redefining the problem

Defining the approximate version of the problem and giving strict bounds on the error in the solution in the crux of the matter in [FM91, Mil89, LM90, For92]. Milenkovic [Mil89] shows how to calculate approximate curve arrangements using rounded-arithmetic. He substitutes each curve by a piece-wise differentiable curve which differs from the original curve by not more than the specified bound. [LM90] presents an algorithm for finding the ϵ-strongly convex hull. A polygon is ϵ-strong convex if it remains convex even after its vertices are arbitrarily perturbed by as much as ϵ. In [FM91], lines are replaced by pseudolines, to obtain a numerically stable algorithm for line arrangements. Pseudolines are unbounded, x-monotonic curves. [For92] gives an algorithm for approximate 2D Delaunay triangulation, defined as the triangulation in which the circumcircle of every triangle is *approximately empty*. In fact, the circumcircle is made of three arcs that closely approximate a circle. Milenkovic, in [Mil88] and [Mil93], obtains robust algorithms for line arrangements by substituting the lines with polygonal curves which stay within some specified distance of the original line.

2.2.2 Specifying minimum features

A common feature of many algorithms is the pre-conditioning of the input i.e. imposing the restrictions of minimum feature size and minimum feature separation. Thus, no two points can be closer to each other than ϵ (separation) or an edge has a width ϵ (size). The constraints can affect various geometric entities and the relations between them viz. points, edges, faces, angles. The process of pre-conditioning, ensures that unambiguous, incidence and adjacency relations hold between the geometric entities. This fact is used in [SS84, Mil88, HHK88]. Algorithms for pre-conditioning the input are given in [SS84] (consolidation) and [Mil88] (data normalization), however the time complexities are not stated.

2.2.3 Avoiding redundant evaluations

In many cases the value of a predicate, that evaluates to an uncertainty can be inferred from the values of the previously evaluated predicates. Fortune has coined a term *parsimonious* algorithms to describe algorithms that "ask no stupid questions" [Knu92]. Knuth [Knu92] gives parsimonious algorithms for finding the convex hull. Hoffmann [Hof89] presents an algorithm for the intersection of two simple polygons, which avoids redundant computations. This approach however does not avoid failure in all cases though it is very practical.

2.2.4 Dynamically changing the input perturbation

Using backward error-analysis we can attribute the error in the solution to a small perturbation of the input. Thus the solution that is obtained can be considered as the exact solution of a slightly perturbed input. With this understanding the approaches stated in [Mil93, Mil88, HHK88, SS84] can be thought of imposing an *a priori* bound on the input perturbation. This bound is fixed. Guibas *et al.* [GSS89] have suggested an approach which is vaguely reminiscent of interval analysis, of dynamically changing the perturbation of the input depending on the errors encountered in the computation. The computation of a predicate returns an interval $e = (e.lo, e.hi)$ such that the input is at least $e.lo$ and at most $e.hi$ away from satisfying the predicate. Ambiguities are resolved by increasing the perturbation. This increase is just enough to resolve the ambiguities. Further generalization of this approach is found in [FBZ93]. Other related work is [DS93] where backward error analysis is applied in the presence of uncertain data.

2.3 Summary

Relying solely on input data can lead to complicated issues of geometric realizability i.e. it becomes difficult to ensure that the logical relations as depicted by the symbolic data are satisfiable in the Euclidean space. The precomputed error bounds are very

pessimistic. In the case of dynamically changing input perturbations, upper bounds on the input perturbation are not known even in the simple cases.

The classification of robust algorithms given here is not very strict and more than one idea can be found in any given paper. The review presented here shows that achieving robustness is computationally expensive and thus emphasizes the fallacy of the asymptotic time complexity in the context of geometric algorithms.

Chapter 3

Planar Point Location - An Implementation

Planar point location is one of the fundamental problems in computational geometry. Here we consider the implementation of an optimal algorithm for planar point location in two dimensions given in [EGS86]. The intention to implement the algorithm was to gain insight into the robustness problem. The problems and anomalies encountered in the implementation along with their causes and solutions to overcome them are presented after the algorithm.

3.1 Optimal planar point location

In the two-dimensional case we are given a subdivision of the plane into two or more regions and then asked to determine which of those regions contains a given query point. If the subdivision has many regions and is to be used for a large number of queries the query time can be reduced by preprocessing the subdivision. We describe here an algorithm that is optimal in all respects; $O(lg(n))$ query time, $O(n)$ storage space, and $O(nlg(n))$ preprocessing time. We begin with the introduction of some definitions which can be found [EGS86]. The pattern and the language for elucidating the algorithm is essentially the same as given in [EGS86].

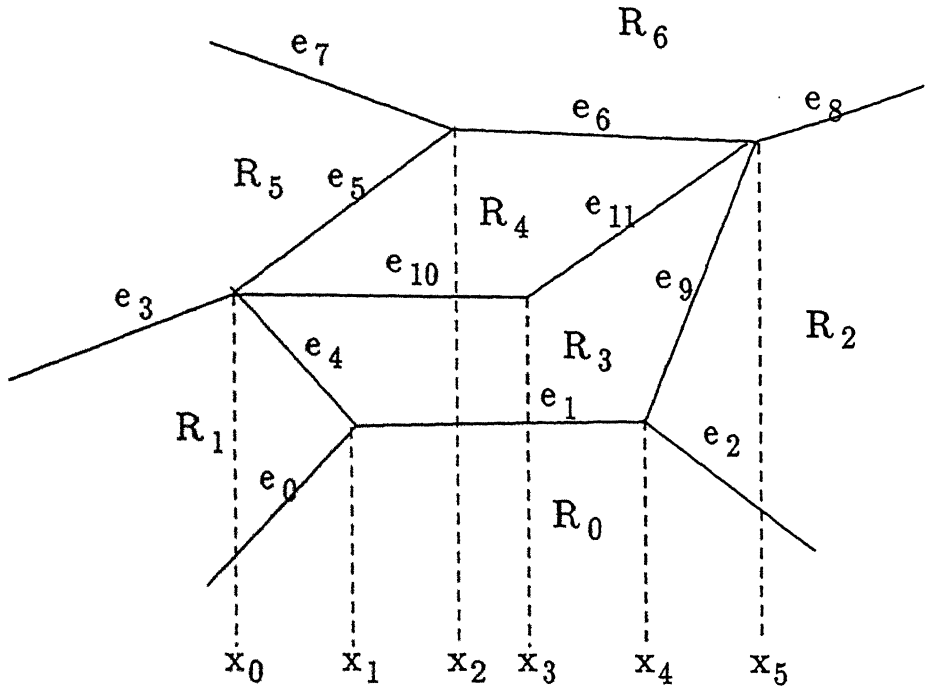


Figure 3.1: Monotone Subdivision

3.1.1 Monotone polygons and subdivisions

An *interval* is a convex subset of a straight line i.e., the whole line, a ray, a segment, a single point, or the empty set. An interval is *proper* if it contains more than one point, and is *open* if it does not contain its endpoints (if any). A subset of the plane is said to be *monotone* if its intersection with any line parallel to the Y-axis is a single interval (possibly empty).

A polygon is defined as an open, connected, and simply connected subset of the plane whose boundary can be partitioned into finitely many points (vertices) and open intervals (edges). With this definition, a polygon may have infinite extent. A subdivision is a partition of the plane into finite number of disjoint polygons (regions), edges, and vertices.

A subdivision is said to be monotone if all its regions are monotone and it has no vertical edges. Also, a subdivision with no vertical edges is monotone if and only if every vertex is incident to at least two distinct edges, one at its left and one at its right. The process of making an arbitrary subdivision monotone is called

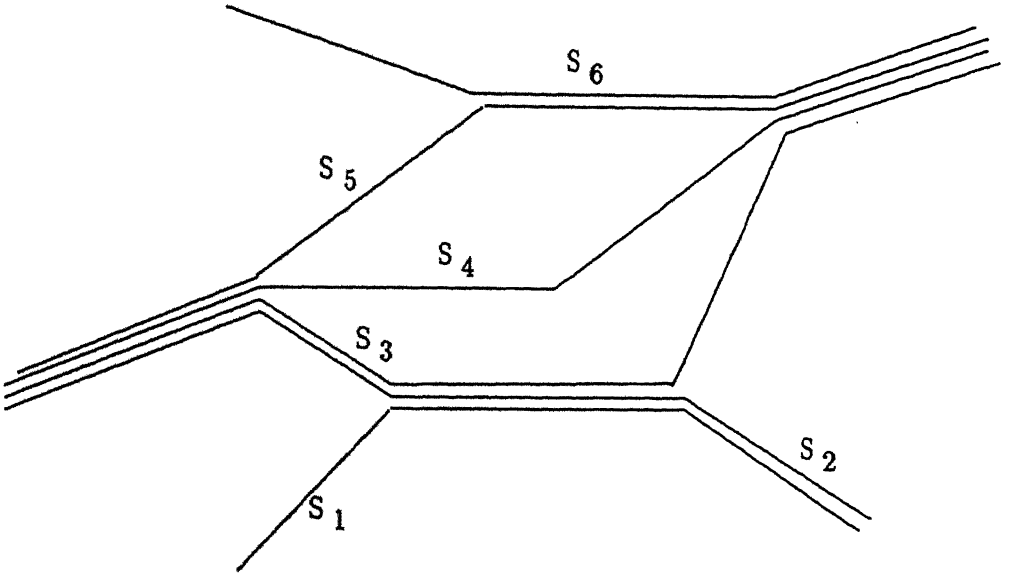


Figure 3.2: Complete Family Of Separators

regularization. A vertex is regular if it has at least one left-going and one right-going edge. The algorithm for regularizing a subdivision can be found in [EGS86, PS85].

3.1.2 Vertical ordering and complete family of separators

We now introduce the relation which vertically orders the subsets of the plane. Given two subsets, A and B, we say that A is above B ($A \gg B$), if for every pair of vertically aligned points (x, ya) of A and (x, yb) of B we have $ya \geq yb$, with strict inequality holding at least once. In this case we also say that B is below A ($B \ll A$). The relation $\gg$ is acyclic.

A separator for a subdivision S is a polygonal line s, consisting of vertices and edges of S, with the property that it meets every vertical line at exactly one point. Since s extends from $x = -\infty$ to $ty = +\infty$, any element of the subdivision that is not part of s is either above or below it.

A complete family of separators for a monotone subdivision S with n regions is a sequence of $n-1$ distinct separators $s_1 \ll s_2 \ll \ldots \ll s_{n-1}$. The complete family of separators for the subdivision in Figure 3.1 is shown in Figure 3.2 There is exactly one region between any two consecutive separators, and also one below s_1, and one

above s_{n-1}. Every monotone subdivision admits a complete family of separators. if a region is below a separator s_i, it must also be below any separator s_j with $j > i$. If a subdivision admits a complete family of separators, its regions can be enumerated as $R_0, R_1, ...R_{n-1}$ in such a way that $R_i \ll s_j$ if and only if $i < j$, in particular,

$R_0 \ll s_1 \ll R_1 \ll s_2 \ll ... \ll s_{n-1} \ll Rn - 1$.

3.1.3 Suboptimal point location algorithm

If the query point p, is below the top-most separator and above the bottom-most separator, by finding the two separators between which the point lies, we can find the region that contains it. We use binary search to find the corresponding separators, since the separators are ordered vertically. This forms the outer loop.

We need an inner loop to find whether p lies above the separator or below it. Using binary search, we find the edge or vertex of the separator whose horizontal projection on the X-axis contains the abscissa of p. By testing p against that edge, we know whether the query point is above or below the separator. While testing p against an edge, it is assumed that the edge contains only its right endpoint but not the left.

The Figure 3.3 shows an infinite, complete binary search tree, **T**, which is used as a flowchart for the outer loop. The convention is that each internal node i represents a test of p against the separator s_i and each leaf j represents the output "p is in R_j".

If each separator is independently stored in a linear array, with all its edges and vertices then we may require more than linear storage. But with a slight modification to the query algorithm, we can ensure that an edge need only be stored in the first separator containing it that would be encountered in a search down the tree **T**. This may give rise to *gaps* between successive stored edges of the separator. The ordered list of stored edges and gaps corresponding to separator s_k is termed as chain c_k. The Figure 3.4 shows the way chains are stored in the data structure. In the figure each node is shown to hold an edge or gap and the corresponding x-values.

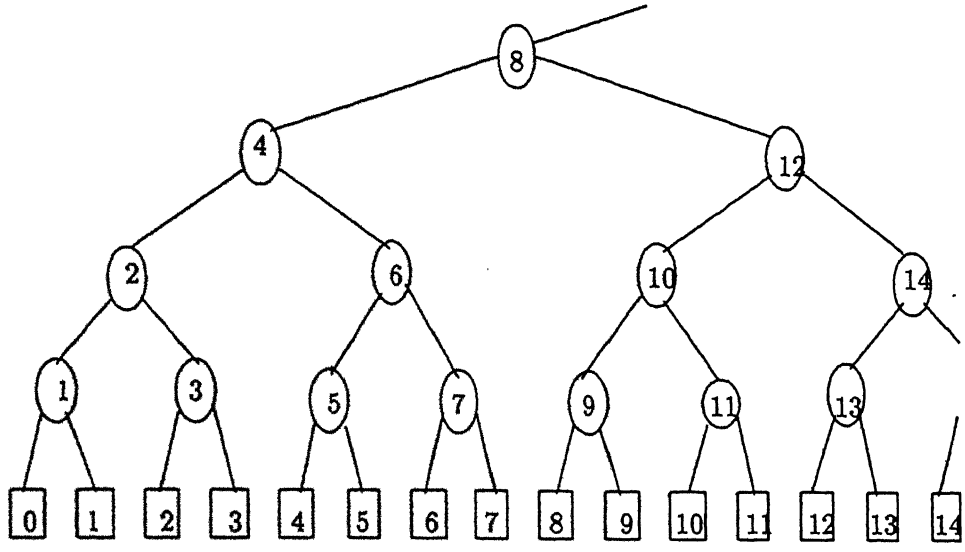


Figure 3.3: The Tree **T**

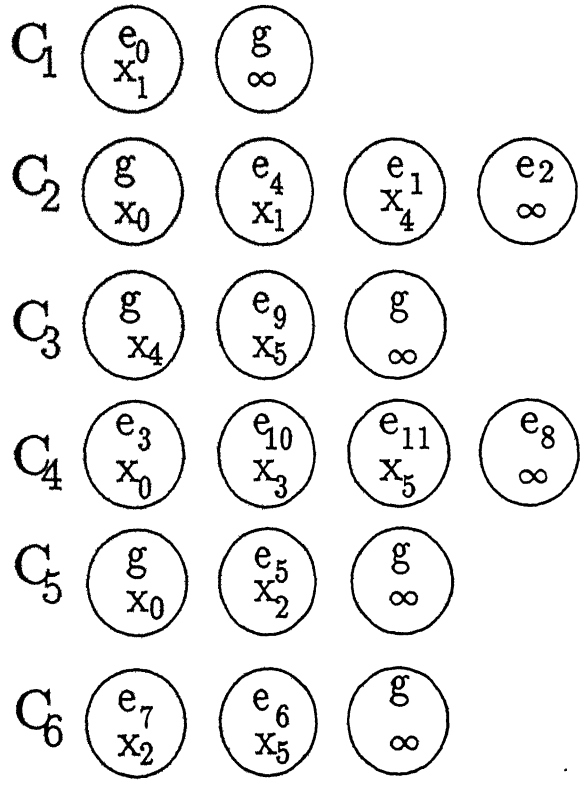


Figure 3.4: The chains

3.1.4 Faster point location

When we discriminate a point against a chain c_k, we must localize it in the x coordinate to within an edge or a gap of c_k. When we continue the search down some child of k, we start this localization process all over again. Edelsbrunner *et al.* proposed an idea to refine the chains so that the localization of p_x in one chain allows us to do the same localization in its children with only constant extra effort. The refinement produces for each chain c_k a list L_k of x-values, defining a partitioning of the X axis into x-intervals. Each such interval of L_k overlaps the x-projection of exactly one edge or gap of c_k and at most two x-intervals of the lists $L_{l(k)}$ and $L_{r(k)}$, where l_k and r_k are left and right children of k, respectively.

The lists L_k and their interconnections are represented by a linked data structure called the *layered dag*. This is a directed acyclic graph whose nodes correspond to tests of three kinds: x-tests, edge tests, and gap tests. A list L_k is represented in the dag by a collection of such nodes: each x-value of L_k gives rise to an x-test, and each interval between successive x-values to an edge or gap test. An x-test node t contains the corresponding x-value of L_k, denoted by $xval(t)$, and two pointers $left(t)$ and $right(t)$ to the adjacent edge or gap nodes of L_k. An edge or gap test node t contains two links $down(t)$ and $up(t)$ to appropriate nodes of $L_{l(k)}$ and $L_{r(k)}$. An edge test node contains a reference to the corresponding edge, $edge(t)$. A gap test node contains the chain number $chain(t) = k$. The layered dag for the subdivision in Figure 3.1 is given in Figure 3.5. In the figure the smaller ovals represent the x-test nodes and the larger ovals represent the edge or gap tests. The continuous lines show the left link for the x-tests and the down link for the gap or edge tests, and the dashed lines show the right link for the x-tests and the up link for the gap or edge tests. The x-values corresponding to the edge or gap test node are shown in figure.

For the algorithm to construct the layered dag, the reader referred to [EGS86]. However there is a small error in that algorithm (Algorithm 3 in [EGS86]). A correction is proposed in *AppendixA* of this thesis.

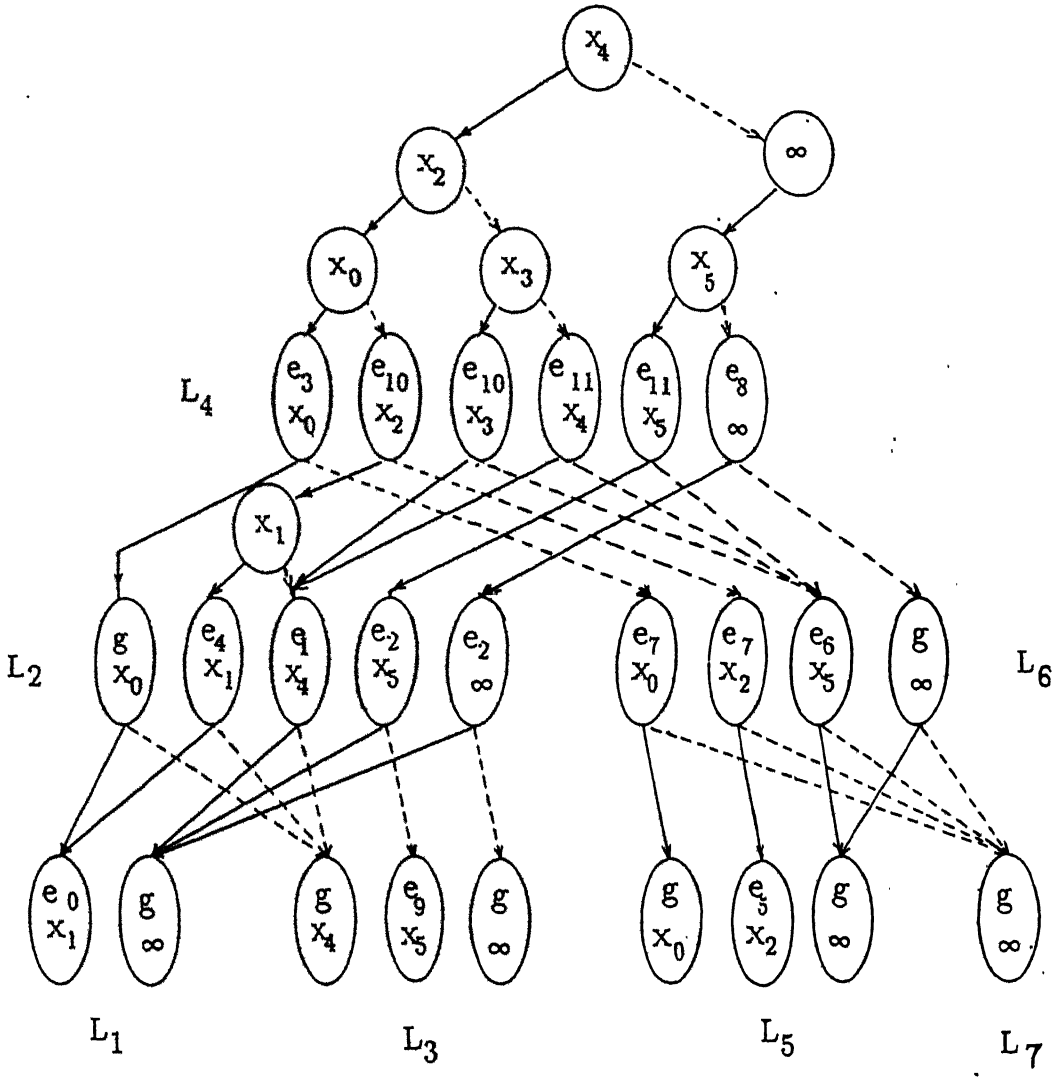


Figure 3.5: The Layered Dag

3.2 Problems and Anomalies

The algorithm was implemented for a graphical input and output. Thus the maximum bit-length of the coordinates is 11, assuming a 1024×1024 window. The coordinates were internally stored as 32 bit integers and the computations were done without approximation. The geometric predicate which tells us whether three points are collinear, clockwise or anticlockwise plays an important role in the 2D planar point location problem. This is represented by the sign of a $3times3$ determinant and the sign can be exactly evaluated when the precision of the computations is twice that of the entries in the determinant. In our case the entries are 11 bit and the computation is done with 32 bits, so the results are exact. Thus the robustness issue never arises. But there are other problems.

3.2.1 The vertical edge assumption

It is assumed in the algorithm that the subdivision has no vertical edges. By trying out different inputs it was found that the probability of having a vertical edge in the subdivision is not negligible. In a practical application vertical edges may be a design feature. The paper [EGS86] does mention that it is possible to remove this assumption with some care. Below we give a method to do away with that assumption. We give the implementation aspects rather than a detailed proof. The proof is based on the symbolic perturbation technique.

The idea is to provide different x-value for each point without violating the planarity of the subdivision. First we sort all points in the subdivision and remove duplicate points. We define the x-value of the ith point P, in the sorted order to be an ordered pair (P_x, i) where P_x in the value of the x-coordinate of P. The $x$value of a point P_i is greater than that of the point P_j if and only if the x-coordinate of P_i is greater than that of P_j or if the x-coordinates are equal than the i is greater than j. By considering an edge with an angle of $\pi/2$ to be a right-going edge and an edge with an angle of $3\pi/2$ to be a left-going edge we can tackle all problems in the algorithms for regularizing the subdivision and finding the complete family of sepa-

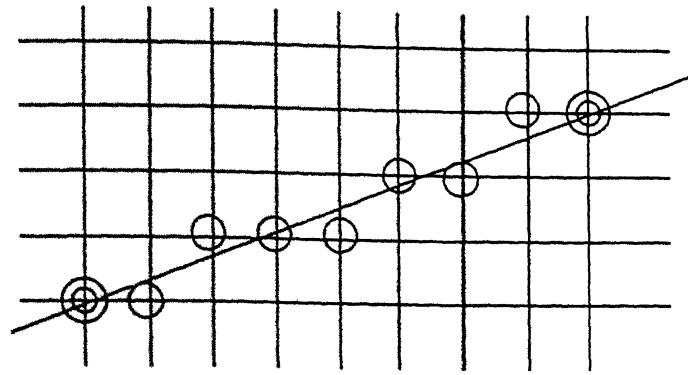


Figure 3.6: Raster Locations for Bressanham's Algorithm

rators. These modifications can be easily implemented and take care of the vertical edge assumption.

3.2.2 Display Anomaly

Even if the mouse is clicked on one of the interior pixels of the edge very rarely is the query point reported to lie on the edge. The reason is that the pixels selected by the line drawing algorithms (refer to [Rog85]) may not always satisfy the equation of the edge as defined by its endpoint. In all line drawing algorithms the displayed pixels lie within a distance of half the pixel-diagonal length of the actual edge. The above anomaly can be corrected by fattening the edge by $1/\sqrt{2}$. This fattening of the edge is analogous to the introduction of uncertainty in the input data and so it restricts our ability to locate the point unambiguously, when two edges are very close to each other. So this modification was discarded later on.

3.3 Summary

Though we have talked only about the point location algorithm throughout this chapter, it cannot be used without prior processing of the graphical input. There is a front end procedure takes the sets of vertices and edges as input and produces a DCEL (refer to [PS85]) for the subdivision. This DCEL forms the input of the point location procedure.

The coordinates of the vertices have relatively small bit lengths and so the computation is essentially exact. When the edges are very close to each other, fattening of the edges gives rise to a situation where the query point may lie on two edges and yet it may not be an endpoint for both of them. The techniques in the next chapter can be applied to resolve this conflict.

Chapter 4

Geometrically Consistent Maintenance of Numerical Data.

At the end of the last chapter we mentioned that the fattening of edges gives rise to some subtle problems in point location. In this chapter we explore the details of that problem and suggest a solution. The most important problem is that the uncertainty in specifying the numerical data gives rise to inconsistent geometric relations amongst the objects. We propose a technique to modify the numerical data so that we do not infer inconsistent relations and thus ensure that an approximated model exists for the relations. The entire discussion in this chapter is limited to 2D and planar geometric objects.

4.1 Introduction

The most common and very specious method to tackle uncertainty in data and computation is the use of *ad hoc* tolerances. Thus the point P is incident on the edge E if the Euclidean distance between P and E is less than ϵ; where ϵ is the *ad hoc* tolerance. Two points are coincident if the Euclidean distance between them is less than ϵ. We illustrate an infeasible situation that can arise due to this method in Figure 4.1*a*. We assume that there is some algorithm that is being executed with the data

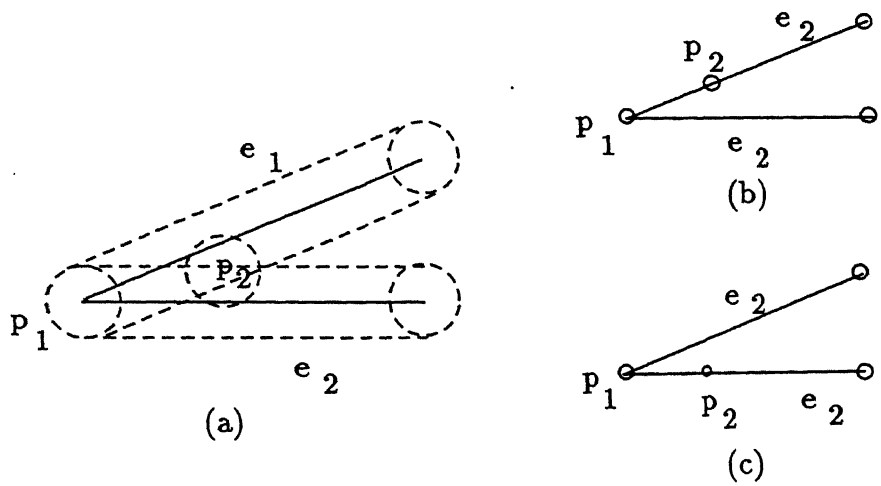


Figure 4.1: Uncertain data and the valid models it admits

in that figure and it needs to compute the incidences that exist amongst the edges and points. At some instant in the execution it finds that the distance between e_1 and p_2 is less than ϵ and so it declares that p_2 is incident on e_1. Later on, after some time it finds that the distance between e_2 and p_2 is less than ϵ. Thus p_2 is incident on both e_1 and e_2 and so e_1 and e_2 intersect at p_2. Still later the algorithm declares that p_1 and p_2 are not coincident because the distance between them is greater than ϵ. Relying solely on the numerical data in the input has given rise to a situation where two edges e_1 and e_2 intersect at two points p_1 and p_2.

To overcome this problem we should move the point p_2 away from e_2 and closer to e_1 as soon as an incidence between e_1 and p_2 is declared. Then the algorithm will not infer incorrectly that the p_2 is incident on e_1. In this chapter we show how to modify the exploit the uncertainty in the input data to maintain geometrical consistency.

There are two geometrically feasible models for the uncertain data shown in Figure 4.1*a*. The models are shown in Figure 4.1*b* and Figure 4.1*c* and both are individually correct but cannot exist simultaneously.

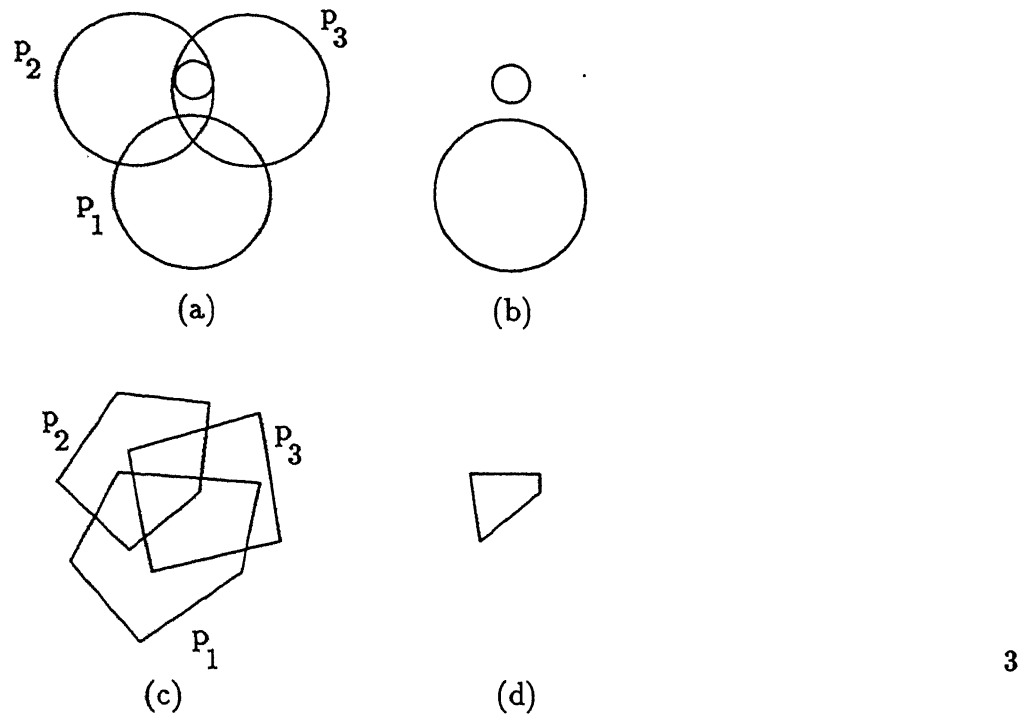


Figure 4.2: Circular and polygonal uncertainty regions

4.2 Polygonal Point

The Euclidean metric has been very extensively used to measure the error in determining the exact position of a point. If the uncertainty in determining the point is at most ϵ then it means that the required point can lie anywhere within a distance of ϵ form the specified point. This gives rise to the circular representation of uncertainty.

We emphasize some problems that can arise due to this representation. A commonly used method [FBZ93] to refine the numerical data is to replace the uncertainty regions of two points by the maximal circle in their intersection, if they intersect, and declare them to be coincident. On the same lines if the uncertainty regions of an edge and a point intersect then the point is replaced by the maximal circle fitting in their intersection. Consider the data as shown in Figure 4.2*a* and we assume that the computation is exact and the uncertainty is due to an error in the input. Now we replace p_1 and p_2 by the maximal circle in their intersection as shown in Figure 4.2*b*. If we call the new point p then p and p_3 are apart, though the circles for p_1,

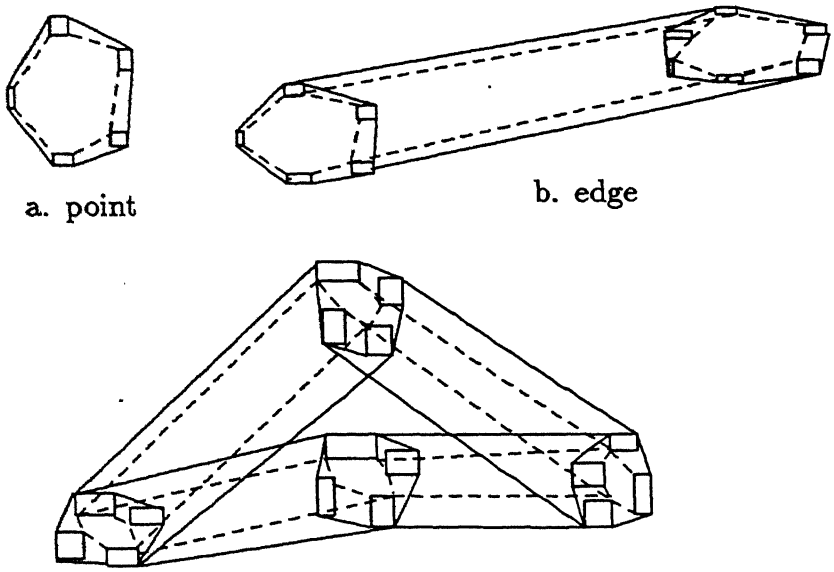


Figure 4.3: Geometric elements in 2D

p_2 and p_3 had a non-empty intersection. This is a severe loss of information. This problem does not arise if the point is convex polygon as shown in Figure 4.2*c* and Figure 4.2*d*.

If the size of the circular uncertainty is decreased to maintain apartness between two points then this reduction is in all directions because the radius has to be decreased. We would like the uncertainty regions to decrease only in that direction which is perpendicular to the line joining the two points by an amount just enough to ensure apartness. Once again the point as a convex polygon can be used to achieve this objective.

Another reason for a polygonal point is the input specification. Almost always the point is specified as an ordered pair of numerical values. Due to errors, each value actually lies in an interval. The uncertainty in the input point is thus a product of two intervals.

4.3 Geometric elements in 2D

A *point* is a polygon whose vertices are *subpoints*. A subpoint is an axis parallel rectangle at a higher precision. The convex hull of the subpoints is called the *outer hull* (H_o) of the point. To define the *internal hull* (H_i) we need to define the *candidate set*. If (H_o) contains only one corner of the subpoint then the diagonally opposite corner is put in the candidate set. If (H_o) contains two corners of the subpoint then the other two corners are put in the candidate set. The internal hull of the point is the largest empty convex subset of the corners in the candidate set. The outer and inner hulls of an edge are the convex hulls of the corresponding hulls of their endpoints. A polygon is a bounded, connected, and simply connected subset of the plane. The polygon boundary is made up of finitely many, alternately occurring points and edges. The H_i s of no two non-adjacent edges of the polygon intersect. The smallest connected subset of the plane that contains the boundary of the polygon is called the polygon exterior and the largest subset of polygon exterior that does not contain the boundary is called the polygon interior. The elements are shown in the Figure 4.3.

As already mentioned, due to the uncertainty in the input data we can represent a point as a product of two intervals. It is assumed that at least the conversion error, which affects the last bit of the mantissa, is present. We convert every numerical value in the input to double precision. The double precision computations will also introduce errors and this is accounted by the rectangular extent of the subpoints. The double precision error e_d is much smaller than the single precision error e_s, so the initial size of the point is much larger than that of every subpoint.

Any point definitely lies inside its inner hull H_i and does not lie outside its outer hull H_o. The hulls can be looked upon as the uncertainty in determining the geometric entities. The ambience defined by the hulls is called the perturbation environment of the objects. The reason we treat the uncertainty as a perturbation will be clear in the section 4.5 .

For a point the hulls can be stored as circular linked lists. Each node in the list stores a floating-point value representing a corner of the subpoint and also a pointer to the corresponding subpoint. The hulls of an edge are also stored explicitly but each

node further holds a pointer to its corresponding point. The hulls of a polygon are implicit, defined by the elements of its boundary. Actually, lot of other information must be stored to facilitate the maintenance of the hulls, for example the list of edges on which the point is incident is stored with every point. We treat the issue of computing and modifying the hulls in section 4.6 .

4.4 Modifying the environment

Our basic aim is to modify the numerical data to achieve geometric consistency. A computation leads to a change in the perturbation environment such that an approximated model exists for all the geometric relations established at the end of this computation. Below we give rules to modify the environment, which are sufficient to test the inclusion of a point in a convex polygon.

- Two points are coincident if their H_i s certainly intersect. In this case the two points are replaced by the intersection of their H_i s. Otherwise the points are forced to be apart by cutting their inner hulls with appropriate straight lines so that the new outer hulls are certainly apart.

- Two points are apart if their H_o s are certainly apart. In case they are not certainly apart and their H_i s are certainly apart then an apartness is enforced by cutting their inner hulls with appropriate straight lines.

- The point P is incident on the edge E if their H_i s certainly intersect and the point P is not incident on any edge that intersects E. If P is incident on E then P is replaced by the intersection of the H_i s of P and E. Otherwise the inner hull of the point is cut appropriately to ensure that the edge and the point are apart.

- Two edges are coincident if their endpoints coincide.

- Two edges intersect if they have no common endpoint and they do not coincide. The point of intersection is the intersection of their H_i s.

- Whenever a point is replaced by its subset, all the edges, of which it is an endpoint are also modified. Recursively, all points incident on these edges are also modified to satisfy the previously established incidences. If any incidence cannot be satisfied then an ambiguity is declared.

- Whenever the size of the inner hull of a point decreases beyond a certain limit (see section 4.5) an ambiguity is declared.

Depending on the geometric properties and relations that the application wants to preserve we can add other rules. If the algorithm does not allow overlapping edges then we can remove them with the rule below.

- Two edges overlap if they have no common endpoint, and both endpoints of one edge are incident on the other edge or one endpoint of either edge is incident on the other edge. In this case we form two or three new edges to replace the two previous edges. The incidences and intersections of the previous edges are maintained by modifying the appropriate objects. In case a particular relation can no longer be satisfied we declare an ambiguity.

If the application requires the intersection of two polygons we can add the following rule.

- Two polygons intersect if their interiors intersect. They cannot intersect uncertainly due to the rules specified above. The new points that arise in the intersection of the polygons are generated by the intersection of the H_i s of the edges of the polygons.

The perturbation environment provides an approximated model for the relations detected by the algorithm. Along with every number the relative error that has occurred in computing it, is stored. There is also a threshold on the relative error to ensure the validity of the numerical results. While computing the intersections of the hulls to find the geometric relation between the objects, errors can occur. When the relative error in the computation is smaller than the threshold we say that the hulls

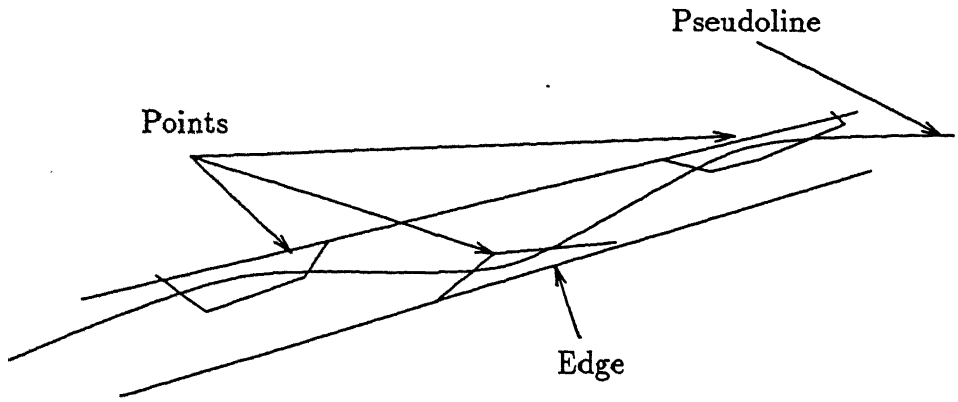


Figure 4.4: The approximate model

either intersect certainly or are certainly apart. If this relative error is more than the threshold then we cannot assure that the hulls certainly intersect or are certainly apart.

The Figure 4.4 explains why the output is said to have an approximated model and not an exact one. In the Figure 4.4 there is no straight line that passes through all the three points that are incident on the edge. We have to replace the line by a pseudoline. The rules ensure that the pseudolines have the desired properties of straight lines.

4.5 Ambiguity Handling

Let e_m be the maximum error that has occurred in the computation of any subpoint of the point P. Whenever the separation between the leftmost and the rightmost corners or that between the bottom-most and the topmost corners in the candidate set is less than $4e_m$ we declare an ambiguity. An ambiguity is also declared when the incidences can no longer be satisfied after the hulls of an edge are shrunk. To resolve the ambiguity we have to expand the point.

Let ϵ be a quantity that is less than the single precision error e_s and much greater than the double precision error e_d. We perturb the centers of the subpoints in the outward direction without losing the convexity of the point. The shape may however change. The Figure 4.5 illustrates the method we employ to expand a convex polygon.

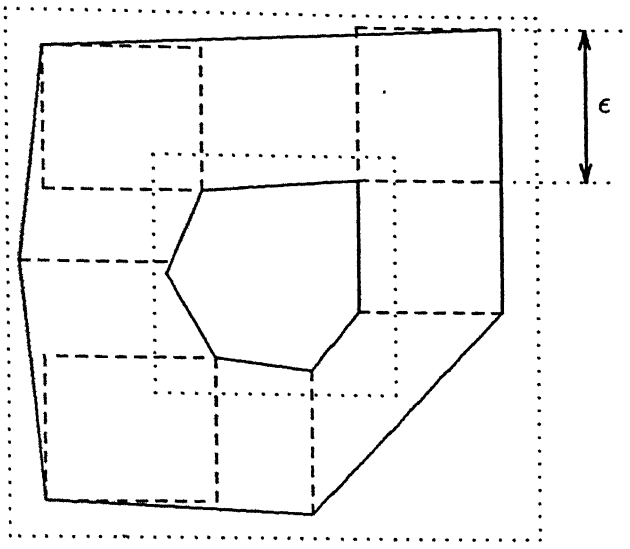


Figure 4.5: Expanding a convex polygon

Successively consider every vertex of the convex polygon to be the origin. In each case find the quadrants that contain no vertices of the polygon. If there is only one such quadrant then move the vertex by ϵ along both the axis in that quadrant. If there are two quadrants then move the vertex by ϵ in the axial direction determined by the two quadrants. If there are three such quadrants then move the point in the middle quadrant. This is equivalent to the perturbation of the input point by not more than ϵ in both the axial directions. In the Figure 4.5 the dashed lines indicate the movement of the vertices of the convex polygon and the two rectangles with dotted lines indicate the original point and the point obtained by perturbing the original point by ϵ in the axial directions.

4.6 Hull maintenance

In this section we focus our attention on finding the hulls of the points and edges. The most important aspect is that we are not allowed to use computations to obtain the hulls because these computations introduce further errors. We should be able to compute the hulls by using simple comparisons of the floating point numbers representing the corners of the subpoints.

Floating point comparison is correct and so we can find the convex hull of two rectangles by using only comparisons. Using this we can compute the inner and outer hulls of the initial point which is a product of two intervals. After that we are never required to find the entire outer hull, we can compute it as the intersection of two convex polygons. The intersection of two convex polygons is always convex and their boundaries may intersect at most two points. Using these two new points we can compute the outer hull of the point and of the edge by using simple comparisons.

The most difficult aspect is finding the inner hull. We were unable to find an algorithm to compute the inner hull that uses only floating point comparisons. There can be more than one hull defined by the corners in the candidate set and we cannot find the largest one without using floating point computations. The basic intention of the inner hull is to ensure that the point definitely lie inside the inner hull despite the errors in the computation of the subpoints. We can be satisfied with any empty convex subset of the candidate set such that it does not intersect with the interior of any subpoint, but which subset will be suitable is not known. The convexity of the inner hull is very important for computing the intersection of hulls.

4.7 Quantification of geometric relations

In some cases a point is incident on more than one edge and yet it may not be their common endpoint. The point must be shrunk so as to lie on only one edge. If no geometric constraint is known then we would like to have a metric to quantify the incidence. We then select that incidence that has maximum measure. We propose one such measure, which is the area of the overlapping portion of the objects. The coincidence C_1 is greater than the coincidence C_2 if the area of C_1 is greater than the area of C_2 or if the areas are equal then the sum of the areas of the objects involved in C_1 is greater than the sum of the objects involved in C_2. A similar order relation holds for incidences. To measure the incidence of the point P on the edge E, only that area of P that lies inside E and avoids every other edge that intersects E. In the most generalized case quantification may involve trying out many possibilities and

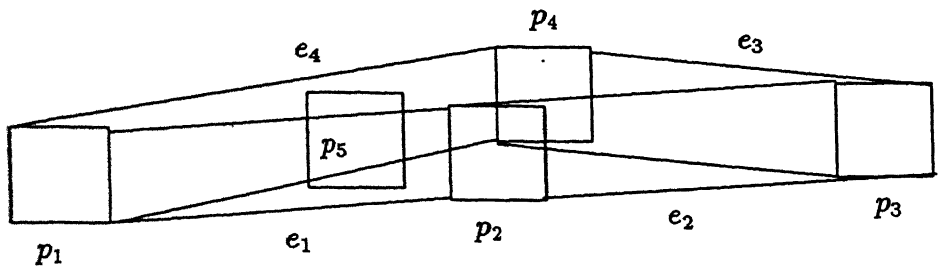


Figure 4.6: Point location in a convex polygon

the selection of the one with the highest measure. Approximate areas can be used without causing any problem to the decision making procedure.

4.8 An illustration

In this section we illustrate the use of the above concepts to locate a point in a convex polygon. We assume that the point location procedure is being used by some other algorithm. That algorithm gives the polygon and the query point to the point location procedure. The point location procedure returns a point if the query point is coincident with any vertex of the polygon, returns an edge if the query point is incident on any edge of the polygon or tells whether the query point is inside or outside the polygon. In all cases the query point is modified to ensure that only the answer returned is true. The higher level algorithm may change the definition of the polygons, later on, when it detects incidences between the polygons edges and vertices. The answer returned by the point location procedure should remain valid even after that.

To locate the point in the polygon we first check whether the query point is coincident with any of its vertices. Then we check whether it is incident on any edge of the polygon. Finally we check whether the point lies inside or outside the polygon.

The Figure 4.6 shows the data that the higher level algorithm sends to the point location procedure. Using the quantification of incidences the procedure infers that the query point p_5 is incident on the edge e_4. The point p_5 is replaced by its area lying entirely in e_4 and avoiding all other edges. Later on the algorithm finds that

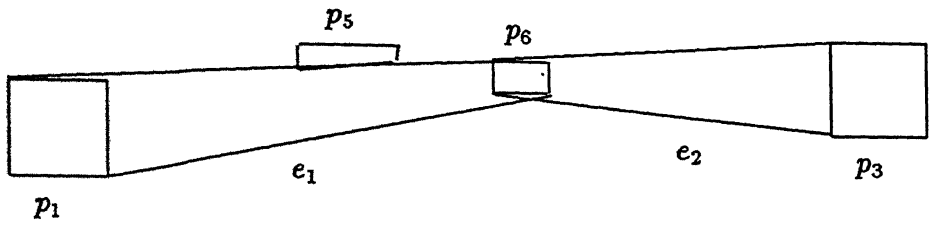


Figure 4.7: Redefinition of the polygon

Figure 4.8: Expansion of the point p_5

the vertices p_2 and p_4 are coincident and so it changes the definition of the polygon. The polygon now has no interior. It is shown in Figure 4.7. The point p_6 is the replacement of p_2 and p_4.

Now the point p_5 does not lie on the edge e_4 (e_4 and e_1 are now same) and so an ambiguity is declared. The point p_5 is then expanded as shown in Figure 4.8.

The Figure 4.9 shows the data that is finally stored. Thus despite the fact that the data is uncertain we have made all decisions correctly and their exits an approximated model that satisfies the relations that the algorithm inferred.

4.9 Summary

We have introduced an interpretation of the uncertainty as a product of intervals which is closer to the way the points are stored. This interpretation of the uncertainty

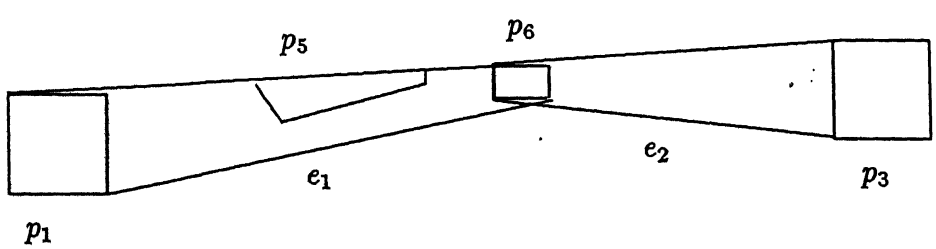


Figure 4.9: Consistent data obtained

is closer to the numerical representation of the objects than the interpretation based on the Euclidean metric. This is especially true when the object is represented by its boundary i.e. the edges and faces are given as symbolic relations and the basic building blocks are the vertices. The vertices are specified by their coordinates.

The technique to quantify incidences is preliminary and yet sufficient to tackle the point location problem. It can be well integrated with the technique to make the numerical data consistent. One of the major problems with the method suggested for maintaining the consistency of the numerical data is that computation of the inner hull. A suitable redefinition of the inner hull may help to overcome this problem.

Chapter 5

Conclusions

It may sound paradoxical that we have to conclude when we are just beginning to understand the robustness of geometric issue. The second chapter presents a unique review of the existing literature. The difficulty of the problem can be comprehended from the fact that very little has been achieved in the long past of this issue. We presented a technique for maintaining the consistency of the numerical data in the fourth chapter. The inefficiency of the method emphasizes that achieving robustness is computationally expensive. Display oriented output presents another different problem, of display inconsistency, which was briefly touched upon at the end of the third chapter. Here we mention some possible directions for future work.

5.1 Efficiency of robust algorithms

Though it was emphasised in the second chapter that the cost of exact computation is exorbitant their is no harm in using it if one is satisfied with the time it consumes. But we rarely need exact answers. A small error in the solution is closer to reality. The most significant aspect of geometric computations is the feasibility of the geometric relations. In the last chapter we demonstrated the use of higher precision to get robust algorithms. The crux of the matter is to have just enough precision to gain a handle on the errors. The approach suggested does not guarantee the existence of

an approximated model with a given error and given precision, but, if it exists it is reported.

The quality of the solution returned by the algorithm is defined by the error it assumes. A method that does not guarantee geometric feasibility is not an algorithm at all. we need to establish a relation between the time taken, the precision required and the error reported. It is possible that with a different heuristic we can get better error bounds with the same precision. This can form the basis for the comparison of robust algorithms. But we have not reached a stage where we have many algorithms to compare. Also a robust algorithm should not be significantly different at the higher level, from its analytic version, so that the programmers life is made a little easy.

5.2 Future Directions

Once again there are two areas that need to be considered, the ones mentioned in the second chapter. The approach given in [Ma93] is very promising because that does not advocate an increase in precision but attaches a sense of reliability to the arithmetic computations. We now need an efficient hardware not just to get the numerical values but also the reliability of those values. On the other hand research should continue to put the issue of geometric feasibility on a sound platform.

Authors tend to make arbitrary assumptions about the data and the computation. These assumptions should be general enough to hold true on any machine. Here we suggest some that are true for most machines.

- We specifically use floating-point numbers or integers, i.e. fixed-point numbers are ruled out. With floating-point numbers we use the relative error model because that provides a better understanding of the errors in floating point numbers. Reasons for insisting on floating-point numbers can be found in [Kn69]. While using integers, an explicit count of the elementary operations should be given. In case the input is assumed to be in single precision we may also assume that the hardware provides support for double precision but not higher than double precision.

- All floating-point numerical values have a bounded relative error. Thus the actual value lies in a definite interval.

These two assumptions are fairly general and the robust algorithm need not assume anything more.

Currently the robustness issue has been extensively tackled only in 2D. Good solutions still elude the problems in 3D. Also, most of the solutions are for planar objects, as the one given in chapter 4. Curves and curved surfaces pose an added difficulty due to the fact that most of the time they are represented by piece-wise linear approximations. A through study of the approach involving the perturbation environment is likely to evolve as simple and fairly general solution. But we cannot give bounds on the final perturbation that the algorithm may return, even in the simple case.

5.3 Summary

The robustness of geometric algorithms is very much an open issue. It is relevant only when the application demands it. The major difficulty in the evolution of a general framework for robustness is that the degree of robustness required, is a very application sensitive issue. In many cases, once in a while, a geometrically infeasible solution is tolerable. But whenever a feasible solution is required it may not be easily obtainable. Moreover, verifying the geometric feasibility of the solution, itself is a very difficult task.

One good thing about robustness is that if you don't want it, you don't have to do anything !

Bibliography

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[BMS] C. Burnikel, K. Melhorn, and S. Schirra. On degeneracy in geometric computations. *to appear*, pages 1–8.

[CDR92] J. Canny, B. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. of the 8th Annual ACM Symposium on Computational Geometry*, pages 251–260, 1992.

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. of the 33rd Annual IEEE Symposium on the Foundations of Computer Science*, pages 387–395, 1992.

[DS88] D. Dobkin and D. Silver. Recipes for geometry and numerical analysis - part i: An empirical study. In *Proc. of the 4th Annual ACM Symposium on Computational Geometry*, pages 93–105, 1988.

[DS93] H. Desaulniers and N. F. Stewart. Robustness of numerical methods in geometric computation when problem data is uncertain. *Computer-Aided Design*, 25(9):539–545, 1993.

[EC92] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In *Proc. of the 8th Annual ACM Symposium on Computational Geometry*, pages 74–82, 1992.

[EGS86] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal of Computing*, 15(2):317–340, 1986.

[EJ91] I. Emiris and J. Canny J. An general approach to removing degeneracies. In *Proc. of the 32nd Annual IEEE Symposium on the Foundations of Computer Science,San Juan*, pages 405–413, 1991.

[EM90] H. Edelsbrunner and E. P. Mucke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, 9(1):66–104, 1990.

[FBZ93] S. Fang, B. Bruderlin, and X. Zhu. Robustness in solid modelling : A tolerance-based intuitionistic approach. *Computer-Aided Design*, 25(9):567–576, 1993.

[FM91] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proc. of the 7th Annual ACM Symposium on Computational Geometry*, pages 334–341, 1991.

[For92] S. Fortune. Numerical stability of algorithms for 2d delaunay triangulations. In *Proc. of the 8th Annual ACM Symposium on Computational Geometry*, pages 83–92, 1992.

[FW93] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. of the 9th Annual ACM Symposium on Computational Geometry*, pages 163–172, 1993.

[GSS89] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: Building robust algorithms from imprecise computations. In *Proc. of the 5th Annual ACM Symposium on Computational Geometry*, pages 208–217, 1989.

[GY86] D. H. Greene and F. F. Yao. Finite-resolution computational geometry. In *Proc. of the 27th Annual IEEE Symposium on the Foundations of Computer Science*, pages 143–152, 1986.

[HHK88] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Towards implementing robust geometric computations. In *Proc. of the 4th Annual ACM Symposium on Computational Geometry*, pages 106–117, 1988.

[Hof89] C. M. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann, 1989.

[KLN91] M. Karasick, D. Lieber, and L. R. Nackman. Efficient delaunay triangulation using rational arithmetic. *ACM Trans. Graphics*, 10(1):71–91, 1991.

[Knu69] D. E. Knuth. *The Art of Computer Programming. Volume II: Seminumerical Algorithms.* Addison-Wesley, 1969.

[Knu92] D. E. Knuth. *Axioms and Hulls.* Springer-Verlag, 1992.

[LM90] Zhenyu Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In *Proc. of the 6th Annual ACM Symposium on Computational Geometry*, pages 235–243, 1990.

[Mas93] G. Masotti. Floating-point numbers with error estimates. *Computer-Aided Design*, 25(9):524–538, 1993.

[Mil88] V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelegence*, 37:377–401, 1988.

[Mil89] V. Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proc. of the 5th Annual ACM Symposium on Computational Geometry*, pages 197–207, 1989.

[Mil93] V. Milenkovic. Robust polygon modelling. *Computer-Aided Design*, 25(9):546–566, 1993.

[OTU87] T. Ottmann, G. Thiemt, and C. Ullrich. Numerical stability of geometric algorithms. In *Proc. of the 3th Annual ACM Symposium on Computational Geometry*, pages 119–125, 1987.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, 1985.

[Rog85] D. F. Rogers. *Procedural Elements of Computer Graphics.* McGraw-Hill, 1985.

[Ros90] H. Rosenberger. *Degeneracy Control in Geometric Programs.* PhD thesis, University of Illinois at Urbana-Champaign, May 1990.

[SS84] M. Segal and C. H. Sequin. Consistent calculations for solids modeling. In *Proc. of the 1st Annual ACM Symposium on Computational Geometry,* pages 29–38, 1984.

[Sto88] J. Stolfi. *Primitives for Computational Geometry.* PhD thesis, Stanford University, May 1988.

[Sug89] K. Sugihara. On finite-precision representations of geometric objects. *Journal of Computer and System Sciences,* 39:236–247, 1989.

[Yap] C. K. Yap. Towards exact geometric computation. *to appear,* pages 1–17.

[Yap88] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. In *Proc. of the 4th Annual ACM Symposium on Computational Geometry,* pages 134–142, 1988.

Appendix A

Correction to the DAG Construction Algorithm

We begin with giving the correct algorithm and then point out the problem with the original one. The entire algorithm is the same as that in [EGS86].

ALGORITHM 3. *Construction of the layered dag.*

1. Set $i \leftarrow 1$. While $i < n$ do:

{*Construct one more level of the tree* **T**. *The first node in this level is* i *and the difference between successive nodes is* $2i$. *A node* k *on this level is the common ancestor of the leaves* $k - i$ *through* $k + i - 1$; *its children (if* $i > 1$*) are the internal nodes* $k - i/2$ *and* $k + i/2$.}

2. Set $k \leftarrow i$. While $k - i < n$ do:

{*Create the list* L_k *from the chain* c_k *and the lists* $L' = L_{l(k)}$}

3. If $i = 1$, set $L' \leftarrow \Phi$, else set $L' \leftarrow L_{k-i/2}$.

****4.** If $i = 1$ or $k + i/2 \geq n$, set $L'' \leftarrow \Phi$, else set $L'' \leftarrow L_{k+i/2}$.

{*This statement is as given in the paper. The inequality is incorrect* }

4. If $i = 1$ or $k + 1 > n$, set $L'' \leftarrow \Phi$, else set $L'' \leftarrow L_{k+i/2}$.

{*Corrected condition* }

5. If $k \geq n$, let c_k be a single gap from $x = -\infty$ to $x = +\infty$.
Split the edges and gaps of c_k at every other x-value of L' and L''.

6. Set $L_k \leftarrow \Phi$. For each edge or gap in c_k, do:

7. to L_k an edge or gap test t representing e.
Set edge(t) $\leftarrow$ e or chain(t) $\leftarrow k$, as appropriate.

8. $L' = \phi$, let down(t) $\leftarrow$ *nil*.
Else, if π_e overlaps only one x-interval of L', let down(t)
point to the corresponding edge or gap test of L'.
Else, π_e overlaps two-intervals of L'; create a new x-test node t'
that chooses between the two corresponding edge or gap tests of L',
and let down(t) $\leftarrow t'$.

9. Set up(t) to *nil*, to an edge or gap test of L'', or to a new
x-test that chooses between two tests of L''.

10. Set $k \leftarrow k + 2i$.

11. Set $i \leftarrow 2i$.

12. Let $r \leftarrow \text{lca}(0, n-1)$. Construct a binary tree of x-tests for the list L_r, and let *root* point to this tree.

Instead of giving an elaborate proof, that the condition in step 4 of the algorithm as published in the paper, constucts an incorrect dag, we give an example to emphasize the fallacy. Let there be 10 regions, $R_0, R_1, \ldots, R_9$. The construction proceeds correctly until $i = 8$. When $i = 8$; $k = 8$ we reach the root L_8. Now $k + i/2 > 10$ and so L_{12} is initialised to Φ despite the fact that it is not empty. Thus there is no connecton left between the root L_8 and the regions R_8 and R_9. These two regions are thus unreachable in the tree, when it is constructed with the algorithm as given in the paper.

The correction is based on the fact that the right child $L_{k+i/2}$ of L_k will be non-empty if the left-most descendent of $L_{k+i/2}$ is non-empty. The left-most descendent

of $L_{k+i/2}$ is L_{k+1} and it is non-empty if $k+1 \leq n$. Thus $L_{k+i/2}$ should be initialised to Φ whenever $k+1 > n$.